Interfacing Hugs and COM

William Javorcik and John Peterson

Research Report YALEU/DCS/RR-1144
~~December 1997~~
January 1998

Yale University

Department of Computer Science
Research Report YALEU/DCS/RR-1144
William Javorcik and John Peterson

Interfacing Hugs and COM

CS690 Independent Project

Interfacing Hugs and COM

Abstract

The goal of this project is to encapsulate the Haskell language interpreter Hugs by a COM wrapper to create a local server version of the Hugs. The task involves designing an interface that would allow exploiting the functionality of Hugs from different clients.

The paper is divided into three parts. The first part provides fundamentals of the Component Object Model and briefly covers issues closely related to the component model of programming in general. The second part describes what has been done in detail and looks at how different issues were implemented. The third part provides guidelines about the parts of the project that have been left open to be implemented in future.

Component Model of Programming

The component model of programming is relatively a new trend in programming methodologies. This section describes its evolution and individual features in general.

Evolution of Components

The traditional way of writing a program has been to write it as a big monolith. Every part of the program knew about all other parts and could call functions or access data in any other part. Evolution in programming methodologies always tended to break up the programs into smaller units that are easier to manage. Modular programming restricted the access to some parts of the program and introduced the concept of lifetime and scope of variables and functions. Object oriented programming encapsulated certain areas of the program's functionality into objects and further restricted direct access to data.

A similar trend towards breaking up the program into smaller and more manageable units could be seen at the executable file level. Originally, applications were stored in individual executable files. Later on, dynamically linked libraries and shared modules were introduced to allow for sharing the functionality between individual programs.

The natural progression of simplifying the programming was to break up the program into smaller components. The components should be relatively independent of each other and, just like objects, they should provide well-defined functionality that can be easily used by other components or parts of the program. The difference is that the components are executable units that can be utilized on their own.

Writing programs consisting of components can also be looked at as a natural combination of the object oriented programming and sharing executable program segments. It allows for isolating parts of functionality into separate executable units that communicate with the world outside through well-defined interfaces. This concept is very similar to the concept of objects. The fact that the components are executable units has many profound consequences and significantly impacts the application development process. It can also provide great benefits to the application programmer in the long run.

Component Object Model (COM)

COM or **C**omponent **O**bject **M**odel is the Microsoft Corporation's implementation of writing programs consisting of components. There are other similar technologies supported by different players in the industry, such as JavaBeans, or CORBA.

These technologies have many things in common. Microsoft's COM is just one way of resolving the issues and defining standards which make the whole concept possible. The following sections look at these issues in detail.

Client-Server Architecture

The Component Object Model is based on the client-server architecture. A component is said to be a **server**, i.e. a provider of **services**, if other programs **connect** to it and request its services. The programs connecting to the server are called **clients**. A client of a server may be a server to other clients. It is also possible for two components to be both clients and servers at the same time.

Different Kinds of Servers

COM supports distributed computing by allowing three different kinds of servers:

- DLL servers
- Local servers
- Remote servers

DLL servers are implemented as DLLs (**D**ynamic **L**ink **L**ibraries). A DLL gets loaded into the client's process address space when the connection to the server is made. There is no mechanism to unload a particular server, but it is possible to unload all unused DLLs.

Local servers are implemented as executable files and they run as separate processes on the same machine as the client process. The server process is usually started automatically by the COM subsystem on the client's behalf. The server process usually terminates once all connections to it are closed, but there is a possibility to lock the server to prevent it from unloading itself.

Remote servers run as separate processes on a machine different from the client machine. **D**istributed **COM** (DCOM) must be configured on both machines in order to use remote servers.

The DLL servers are also referred to as **in-process** servers. The local and remote servers are known as **out-of-process** servers. A client can communicate with any type of server without having to be modified.

Calling Services and Passing Parameters and Results

Calling services and passing parameters to in-process servers is easy since both the client and the server share the same address space.

Calls to local servers are issued using the LPC (**L**ocal **P**rocedure **C**all) mechanism. The parameters must be **marshaled** across the process boundary but they need no conversion to a standard format.

Calls to remote servers are issued using the RPC (**R**emote **P**rocedure **C**all) mechanism. Marshaling the parameters and results requires putting them into the standard format and packetizing them for the transport across the network.

Relation of Objects and Components

The object-oriented technology fits the component model very well. This is why C++ is the language of choice for implementing COM components.

A component corresponds to an object that exposes multiple interfaces. Since a C++ class cannot have multiple interfaces, a component is usually implemented using multiple classes.

Primary Issues

The Component Object Model was designed with certain requirements in mind. These requirements actually constitute the benefits that COM provides for the developer. As shown below, many of these properties are closely related.

Independence from the Source Code

Individual components must be usable without access to their source code. All functionality of a component must be **published**, i.e., must be specified and known up front.

Consequence:
The components must publish their presence on the system

Communication through Interfaces

All communication with components must be performed through interfaces. **Interfaces** are sets of functions that the component supports. Each interface represents certain area of functionality. A component can support multiple interfaces.

Consequence:
The components must be able to return information about interfaces they support. The components may also publish the interfaces they support on the system. Since the communication is possible only through an interface, the components must share a common interface (`IUnknown`).

An interface is uniquely identified by a 128-bit value called **IID** (**I**nterface **ID**). An IID is an instance of a **GUID** (**G**lobally **U**nique **Id**entifier).

Immutability of Interfaces

The interfaces supported by a component can never change. An interface cannot be modified or deleted after it is published.

An interface is changed if any of the following changes:

- Number of functions in an interface
- Order of functions in an interface
- Number of parameters in a function
- Order of parameters in a function
- Types of parameters in a function
- Possible return values from a function
- Types of return values from a function
- Meanings of parameters in a function
- Meanings of functions in an interface

Versioning of Components

If a change is necessary, a **new** interface that extends the functionality of the previous one must be **added** (this requirement is based on immutability of interfaces). Adding the new interface must not break the functionality of previous interfaces if they share code or data.

Consequence:
The components are likely to support multiple interfaces for the same area of functionality. The client must be able to determine which interface it can use.

Polymorphism of Components

The components implement polymorphism using multiple interfaces. COM does not support polymorphism through inheritance.

Consequence:
The implementation of interfaces is likely to share component's internal data and functions.

Dynamic Linking

A component's functionality must be accessible without having to go through a linking process. Replacing an older component by a newer one must be possible in runtime without impacting the functionality of any clients connected to the component.

Consequence:
The functions in an interface must be callable by address, not by a name. If the functions the component publishes are callable by name, there has to be at least one function callable by address.

Server Location

The location of the server is arbitrary. A client can choose what kind of server it wants to connect to.

Consequence:
The location of the server can only impact the performance, and never the functionality.
The parameters passed to and from a local (remote) server must be marshaled across the process (machine) boundary.

Machine Independence

The components should be machine independent. A client must be able to communicate with a server running on a different platform.

Consequence:
The parameters passed to and from the server must be put to a standard form and converted on to the native form at each end of the communication.

Language Neutrality

It must be possible to write COM programs in arbitrary language.

Consequence:
COM is a **binary** standard. Any programming language that can create binary structures compatible with the COM definition is suitable for writing COM applications.

Important Implementation Issues

Reference Counting

Reference counting is a mechanism that allows the servers to count the references made to its services. The COM specification includes reference counting to free the programmer from the necessity of explicitly unloading the server. This mechanism guarantees that the server will be never unloaded at an improper time.

Marshaling

Marshaling is the process of transporting the parameters of a function from the address space of the client into the address space of the server (and vice versa).

Marshaling between the processes on the same machine involves accessing and copying the data across the process boundary. If the client and the server run on different machines with possibly different architectures, marshaling must put the data into a standard format.

Marshaling is implemented using a so-called **marshaling in-process server** (marshaling DLL). A DLL is used since it can be mapped into the process space of both the client and the server and thus it can provide services on both sides of the communication pipe.

The code handling the marshaling of the outgoing parameters is called **proxy**; the code handling incoming parameters is called **stub**.

Marshaling is implemented by the **`IMarshal`** interface. It is possible to implement customized marshaling to improve marshaling of built-in data types or to support new types.

IDL / MIDL

In order to facilitate easy maintenance of interfaces, Microsoft adopted a language called **IDL** (**I**nterface **D**escription **L**anguage) which was originally a part of the Open Software Foundation's (OSF) Distributed Computed Environment (DCE).

MIDL is the IDL language compiler developed by Microsoft that takes a description of the interfaces in IDL and generates the C/C++ header files containing the class/structure definitions of interfaces. Apart from the interface header files, MIDL also generates the proxy/stub code for a marshaling DLL.

MIDL can also be used to generate the type libraries for the components.

Memory Allocation

In order to handle parameters of arbitrary size in a consistent manner, COM defines that in-out and out parameters must be implemented using pointers. The memory pointed to by the pointers can be allocated by the client and may have to be released by the server (or vice versa).

To deal with the situation that memory will be allocated and released by different components, COM defines a set of memory allocation functions encapsulated in the **`IMalloc`** interface.

Class Factories

The task of creating a COM component is implemented by another component that exists solely for the purpose of creating components. This component is called **class factory** and it communicates with its clients through the **`IclassFactory`** interface.

It is possible to define customized class factories that can perform special tasks when creating components. Such factories must implement the `IclassFactory` interface.

Registry

The implementation of COM heavily relies on the central Windows data repository, otherwise known as the **Registration Database**, or **Registry**. This approach is necessary since a client must be able to connect to a server without knowing anything explicit about it other than the fact that the server exists.

The components are registered in several ways:

1. By the **Class ID** under the `HKEY_CLASSES_ROOT/CLSID` key.
2. By the version-independent **Program ID** in the `<Program>.<Component>` format under the `HKEY_CLASSES_ROOT` key.
3. By the version-dependent **Program ID** in the `<Program>.<Component>.<Version>` format under the `HKEY_CLASSES_ROOT` key.

This system allows an easy lookup of a component by its Class ID or Program ID. It is also possible to register a component as a member of one or more component categories.

For DCOM, additional information in registry is required (such as location of the remote servers).

Aggregation and Containment

Aggregation and containment are two methods used by COM to implement component reuse. Both methods combine two components into a unit in which one component is identified as the **outer** component and the second component is identified as the **inner** component.

In **containment**, the inner component is completely encapsulated by the outer component.

In **aggregation**, the outer component reuses the inner component and extends its functionality by adding to new interfaces that may or may not use the inner component's functionality. This is the preferred method to implement inheritance on the component level.

Threading Models

Threads are an important programming methodology especially when it comes to responsive user interfaces and I/O operations. COM must support threading since the components may be accessing COM servers from different threads.

Windows 32 defines two kinds of threads: **user-interface** threads and **worker** threads. In COM terminology these are known as **apartment** threads and **free** threads, respectively.

Under the apartment model, there can be multiple COM objects on a thread, but calls to an object must execute on the thread that "owns" the object. This thread is called the object's apartment thread and interface requests to the object from another thread must be marshaled to the apartment thread that owns the object. Standard marshaling is used to cross the thread boundaries.

Summary of Benefits and Drawbacks

Portability

Both COM and DCOM natively run on Microsoft Windows NT and Windows 95. It is expected that any 32-bit Windows operating system will be able to run COM and DCOM. Microsoft has also ported COM to the Macintosh platform.

Microsoft has contracted Software AG to port COM to other platforms; both UNIX and non-UNIX based.

Extendibility

COM is a proprietary standard, therefore it is not extensible. Microsoft is talking about making it an open standard, and it is conceivable once COM is being used on other platforms as well.

Speed

In-process server calls are very fast and they incur minimal overhead. Local servers are slower compared to in-process servers since their performance is impacted by the necessity of marshaling the parameters across process boundaries. Remote servers are order of magnitude slower than local servers and their effective use may require special optimizations.

Implementation of the Project

The project itself consists of three separate program units:

- Marshaling DLL
- Local Hugs server
- Hugs Client

About Program Units

All files belonging to a program unit are located in a separate directory.

To build/make a program unit, Microsoft command line development tools were used. Each program unit directory contains a `make` file compatible with the Microsoft `nmake` program.

The intermediate output files (binary files like `*.obj`, `*.res`, `*.map`) are stored in the `TEMP` directory.

To compile a program unit with debug information, enter:

```
nmake
```

To compile a program unit without debug information, enter:
```
nmake nodebug=1
```

To register the program unit (if applicable), enter:
```
nmake register
```

To unregister the program unit (if applicable), enter:
```
nmake unregister
```

To clean up binaries, enter:
```
nmake clean
```

To clean up all generated files, enter:
```
nmake cleanall
```

A program unit is described using the following subsections:
- Functionality
- Invocation
- Source Files (in detail)
- Modifying the Code

What Has Been Done

Sample code from the Platform SDK was used to create the COM framework for all three program units.

A simple interface named **`IHugs`** was defined to communicate with the server version of Hugs. The interface provides basic capability to initialize the Hugs server, load a project into it and examine parts of it internal symbol table.

In order to allow for future extension, support for aggregation of components was included in the framework. A simple aggregating interface named **`IHugsU`** was defined to demonstrate aggregation.

The `IHugsU` interface code was commented out in the source and make files to simplify program development. Otherwise, every change to the `IHugs` interface would have to be implemented in the `IHugsU` interface as well. The functionality of the programs is not impacted by this change since currently there is no immediate need to support aggregation of components.

The `Ihugs` Interface

The `IHugs` interface contains functions that demonstrate the basic functionality that is achievable by the use of COM. The following functions has been defined:
```
HRESULT Test(short nNum, short *pnRes);
HRESULT CInitHugsServer(int iArgc, unsigned char *szArgv);
HRESULT CGetLastError(string255 szErr);
HRESULT CLoadProject(unsigned char *szFileName);
HRESULT CLoadFile(unsigned char *szFileName);
HRESULT CGetModuleCount(int *pnModule);
HRESULT CGetModuleName(int nModule, string255 szName);
```

The Test function is the only function that has been correctly implemented in the `IHugsU` interface. It only serves as a test bed for various COM function issues related to aggregation.

The first four functions deal with generic Hugs functionality:
`CInitHugsServer` initializes the Hugs interpreter (includes reading in the standard prelude).
`CGetLastError` retrieves the textual information about the last error that occurred.

`CLoadProject` instructs Hugs to load a project whose name is the argument of the function.
`CLoadFile` instructs Hugs to load a file whose name is the argument of the function.

The last two functions implement an interface for obtaining information about modules:
`CGetModuleCount` returns the number of modules loaded into Hugs.
`CGetModuleName` returns the name of the module whose number is the argument of the function.

The Hugs Marshaling DLL

Functionality

The marshaling DLL functions as a communication server between the Hugs client and the Hugs server executables. It provides the stub/proxy code for marshaling parameters of the interface functions between the client and the server.

In other words, it is an in-process COM server that offers standard marshaling services for the interfaces it is registered to handle.

Invocation

The marshaling DLL is invoked automatically by the COM subsystem.

Registering the Program

The program is automatically registered using the `register` utility if the make process completes successfully.

Despite the fact that it is a DLL, it is possible to "invoke" it for the purpose of registering and unregistering its functionality. This "invocation" must be, however, done using an executable program – `regsvr32.exe`.

To register the marshaling DLL, enter:
```
regsvr32 marshal.dll
```

To unregister the marshaling DLL, enter:
```
regsvr32 /u marshal.dll
```

Source Files

The source files are located in the `marshal` directory.

Source files modifiable by the developer:

`makefile`	the make file
`mihugs.idl`	the IDL interface description file containing descriptions of all Hugs interfaces
`marshal.cpp`	the main implementation file for `marshal.dll`
`marshal.rc`	the resource definition file for `marshal.dll`
`marshal.ico`	the icon resource file for `marshal.dll`

Source files generated by the MIDL compiler:

`mihugs.h`	the interface include file for the specified interfaces.
`dlldata.c`	DLL data routines for the proxies and other functions
`mihugs_i.c`	the data definitions of the GUIDs for the marshaled interfaces.
`mihugs_p.c`	the actual proxy and stub functions for the interface methods.

Interface Description File (`mihugs.idl`)

This file contains the description of the interface implemented by the Hugs server.

Interface Definition in IDL

Here is a section of the file defining the `Ihugs` interface.

```
[uuid(911a1190-685c-11d1-b616-0000c0b95d8e),
 object
]
interface IHugs : IUnknown
{
  import "unknwn.idl";

  HRESULT Test([in] short nNum, [out] short* pnRes);
  HRESULT CInitHugsServer([in] int iArgc, [in, string] char* szArgv);
  HRESULT CGetLastError([out] string255 szErr);
  HRESULT CLoadProject([in, string] char* szFileName);
  HRESULT CLoadFile([in, string] char* szFileName);
  // Module access
  HRESULT CGetModuleCount([out] int* pnModules);
  HRESULT CGetModuleName([in] int nModule, [out] string255 szName);
}
```

The translation of this definition to C++ is as follows (simplified):

```
interface DECLSPEC_UUID("911a1190-685c-11d1-b616-0000c0b95d8e")
IHugs : public IUnknown
{
public:
  virtual HRESULT STDMETHODCALLTYPE
    Test(short nNum, short __RPC_FAR *pnRes) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    CInitHugsServer(int iArgc, unsigned char __RPC_FAR *szArgv) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    CGetLastError(string255 szErr) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    CLoadProject(unsigned char __RPC_FAR *szFileName) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    CLoadFile(unsigned char __RPC_FAR *szFileName) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    CGetModuleCount(int __RPC_FAR *pnModule) = 0;
  virtual HRESULT STDMETHODCALLTYPE
    CGetModuleName(int nModule, string255 szName) = 0;
};
```

The interface name is `Ihugs`; it is derived from `IUnknown`.

The UUID was generated using the `uuidgen` tool, which is a part of the Microsoft Development Studio. The UUID assigned to this interface should not change once the interface was published. It is necessary to assign a unique UUID to each interface.

The `import` section is equivalent to the C language `#include` directive, except that it is possible to import the same file more then once without having to worry about multiply defined symbols.

The function declarations are very similar to C. The `[in]` and `[out]` parameter attributes specify which way the parameter is used and help MIDL generate more optimized code.

The `[string]` attribute instructs MIDL that the following pointer to `char` is really a pointer to a null-terminated string, so that the proxy/stub code that will be generated will marshal the whole string including the zero terminator as opposed to marshaling just one character.

Support for the `IHugsU` interface

The definition of the `IHugsU` interface is commented out.

Main Implementation File (`marshal.cpp`)

This file contains the code implementing the DLL itself. The main entry point to the DLL is the `DllMain` function. Other important functions in this source file are `DllRegisterServer` and `DllUnregisterServer`.

The marshaling DLL is registered as the provider of standard marshaling for the interfaces specified in `mihugs.idl`. Each interface is registered using the GUID specified for it.

Support for the `IHugsU` interface

The code registering and unregistering the `IHugsU` interface is commented out.

Modifying the Code

Enabling the `IHugsU` interface

To enable support for the `IHugsU` interface:

1. `mihugs.idl`: Uncomment the section defining the `IhugsU` interface
2. `marshal.cpp`: Uncomment the Create Registry Entries and the Delete Registry Entries sections for the `IHugsU` interface
3. Erase the temporary binaries in the `TEMP` directory and re-make the DLL

Adding a New Function to an Existing Interface

This procedure can be **only** applied during the development phase of the program when the interface was **not** published yet. To add a new function to an existing interface, a new version of the interface with a new IID must be created.

To add a new function to an existing interface:

1. `mihugs.idl`: Add the function declaration into the interface section
2. Erase the temporary binaries in the `TEMP` directory and re-make the DLL

Adding a New Interface

To add a new interface to the marshaling DLL:

1. `mihugs.idl`: Add the new interface section
2. `mihugs.idl`: Generate a new IID using the `uuidgen` program and paste it into the `uuid` attribute of the new interface
3. `marshal.cpp`: Add the Create Registry Entries and the Delete Registry Entries sections for the new interface
4. Erase the temporary binaries in the `TEMP` directory and re-make the DLL

The Hugs COM Client

Functionality

The Hugs COM client is a separate executable program named `HugsCln.exe`.

After the program is run, it initializes the COM functionality and displays a simple menu. The main menu contains a submenu for each interface supported. Each submenu contains the `Create` and `Release` entries that create and dispose of a component. Additional menu entries correspond to individual interface functions. Each function can print debugging traces into the main window.

When the `Create` function is invoked, the client connects to the `HugsSrv` server and creates the desired component. If the server is not running, it is started automatically. It is possible to create one component for each interface. The Hugs COM server will be started automatically if there are no active components and it will terminate if the number of active components goes to zero.

The `IHugs` Interface Submenu

This submenu contains the following selections:

```
IHugs::Create
IHugs::Release
IHugs::Test
IHugs::Intialize
IHugs::GetLastError
IHugs::LoadProject
IHugs::LoadFile
IHugs::GetModuleCount
IHugs::GetModuleName
IHugs::GetModuleNames
```

The `Create` function connects to the Hugs COM server and creates an object of the type `COHugs`. The `Release` function disposes of the object. If it was the last object housed by the server, it also unloads the server.

The rest of the functions except for `GetModuleNames` invoke the corresponding `IHugs` interface functions. `GetModuleNames` is implemented using both `GetModuleCount` and `GetModuleName`.

Invocation

The program can be invoked by either double-clicking the program's icon, or by entering its name on the command line. The program does not take any command line switches.

Registering the Program

The Hugs client does not need to be registered.

Source Files

The source files are located in the `HugsCln` directory.

Source files modifiable by the developer:

`makefile` the make file
`HugsCln.h` the include file containing class declarations, function prototypes and resource identifiers
`HugsCln.cpp` the main implementation file, containing the `WinMain` and `CMainWindow` implementation, as well as the main menu dispatching
`HugsCln.rc` the resource definition file for `HugsCln.exe`
`HugsCln.ico` the icon resource file for `HugsCln.exe`

Hugs Client Include File (`HugsCln.h`)

This include file defines the main window class as well as the resource identifiers (menu items, strings, etc.). The window class is a class derived from the `CvirWindow` in the `AppUtil` module:

```
class CMainWindow: public CvirWindow;
```

Support for the `IHugsU` interface

The code defining entries for the `IHugsU` interface is commented out.

Hugs Client Source File (`HugsCln.cpp`)

This source file contains all functionality that the Hugs COM client program implements.

The file implements the `CMainWindow` class defined in `HugsCln.cpp` and the `WinMain` function.

Support for the `IHugsU` interface

The code supporting the `IHugsU` interface is commented out.

Modifying the Code

Enabling the `IHugsU` interface

To enable the support for the `IHugsU` interface:

1. Marshaling DLL: Enable support for the `IHugsU` interface
2. `HugsCln.h`: Uncomment the section specifying the menu command identifiers for the `IHugsU` interface
3. `HugsCln.h`: Since the `IHugsU` interface inherits from `IHugs`, it may be necessary to add the menu command identifiers for the new functions that `IHugsU` inherits from `IHugs`
4. `HugsCln.cpp`: Uncomment the `IHugsU` interface pointer declaration in the `CMainWindow::DoMenu` function
5. `HugsCln.cpp`: Uncomment the `IHugsU` interface menu command processing in the `CMainWindow::DoMenu` function
6. `HugsCln.cpp`: Since the `IHugsU` interface inherits from `IHugs`, it may be necessary to add menu command processing for the new functions that `IHugsU` inherits from `IHugs`
7. `HugsCln.rc`: Uncomment the `IHugsU` interface menu definition section
8. `HugsCln.rc`: Since the `IHugsU` interface inherits from `IHugs`, it may be necessary to add menu items for the new functions that `IHugsU` inherits from `Ihugs`
9. Erase the temporary binaries in the `TEMP` directory and re-make the `HugsCln` program

Adding a New Function to an Existing Interface

This procedure can be **only** applied during the development phase of the program when the interface was **not** published yet. To add a new function to an existing interface, a new version of the interface with a new IID must be created.

To add a new function to an existing interface:

1. Marshaling DLL: Add a new function to an existing interface
2. `HugsCln.h`: Add the menu command identifier for the new function
3. `HugsCln.cpp`: Add the menu command processing for the new function
4. `HugsCln.rc`: Add a menu item for the new function
5. Erase the temporary binaries in the `TEMP` directory and re-make the `HugsCln` program

Adding a New Interface

To add a new interface to the `HugsCln` program:

1. Marshaling DLL: Add a new interface
2. `HugsCln.h`: Add the menu command identifiers for all functions in the new interface
3. `HugsCln.cpp`: Add the menu command processing for all functions in the new interface
4. `HugsCln.rc`: Add the menu items for all functions in the new interface
5. Erase the temporary binaries in the `TEMP` directory and re-make the `HugsCln` program

Displaying the Server's About Window

It is possible for the client to request the About Window from the server. The client will try to lookup the server's window by name:

```
HWND hWnd = FindWindow(NULL, TEXT(SERVER_WINDOW_TITLE_STR));
if (NULL != hWnd)
  PostMessage(hWnd, WM_COMMAND, IDM_HELP_ABOUT, NULL);
```

In order for this lookup to succeed, the developer must make sure that both the client and the server know each other's window names correctly:

```
#define MAIN_WINDOW_TITLE_STR      "HugsCln: COM Interface for Hugs"
#define SERVER_WINDOW_TITLE_STR    "HugsSrv: COM Interface for Hugs "
```

Example of a Hugs Session

This example will demonstrate functionality of the `IHugs` interface.

1. Start the Hugs Client is started by entering:
   `HugsCln`
   The client starts up and displays the main menu
2. Invoke `Hugs->IHugs::Create`
   The client connects to the server started by the COM subsystem and creates the object implementing the `IHugs` interface
3. Invoke `Hugs-> IHugs::GetModuleCount`
   The program prints the following message:
   ```
   C: === Hugs Menu: IHugs::CGetModuleCount
   C: --pIHugs->IHugs::CGetModuleNames: returned 0
   ```
   This means that there are no modules defined, since the Hugs server was not initialized
4. Invoke `Hugs->IHugs::Initialize`
   The Hugs server is initialized and the standard prelude gets loaded in
5. Invoke `Hugs-> IHugs::GetModuleCount`
   The program prints the following message:
   ```
   C: === Hugs Menu: IHugs::CGetModuleCount
   C: --pIHugs->IHugs::CGetModuleNames: returned 7
   ```
   After the standard prelude is loaded, there are 7 modules
6. Invoke `Hugs->IHugs::GetModuleNames`
   The program prints the following output:
   ```
   C: === Hugs Menu: IHugs::CgetModuleNames
   C: --pIHugs->CGetModuleName: 0 module's name is 'Prelude'
   C: --pIHugs->CGetModuleName: 1 module's name is 'Trace'
   C: --pIHugs->CGetModuleName: 2 module's name is 'Ix'
   C: --pIHugs->CGetModuleName: 3 module's name is 'Maybe'
   C: --pIHugs->CGetModuleName: 4 module's name is 'IO'
   C: --pIHugs->CGetModuleName: 5 module's name is 'IOExts'
   C: --pIHugs->CGetModuleName: 6 module's name is 'Dynamic'
   ```
7. Invoke `Hugs->IHugs::LoadFile`
   The program prints the following output:
   ```
   C: --Calling pIHugs->CLoadFile d:/cs690/hugs/demos/random.hs
   ```
   The file `random.hs` from the `demos` directory is loaded in
8. Invoke `Hugs-> IHugs::GetModuleCount`
   The program prints the following message:
   ```
   C: === Hugs Menu: IHugs::CgetModuleCount
   C: --pIHugs->CGetModuleNames: returned 9
   ```
   Loading the file `random.hs` increased the number of modules by 2
9. Invoke `Hugs->IHugs::GetModuleNames`
   The program prints the following output:
   ```
   C: === Hugs Menu: IHugs::CgetModuleNames
   C: --pIHugs->CGetModuleName: 0 module's name is 'Prelude'
   ```

```
C: --pIHugs->CGetModuleName: 1 module's name is 'Trace'
C: --pIHugs->CGetModuleName: 2 module's name is 'Ix'
C: --pIHugs->CGetModuleName: 3 module's name is 'Maybe'
C: --pIHugs->CGetModuleName: 4 module's name is 'IO'
C: --pIHugs->CGetModuleName: 5 module's name is 'IOExts'
C: --pIHugs->CGetModuleName: 6 module's name is 'Dynamic'
C: --pIHugs->CGetModuleName: 7 module's name is 'Gofer'
C: --pIHugs->CGetModuleName: 8 module's name is 'Random'
```

The Hugs COM Server

Functionality

The Hugs COM server is a separate executable program named `HugsSrv.exe`.

The program is a registered COM server for the `IHugs` (`IHugsU`) interface(s). The Hugs Server is essentially a server version of Hugs wrapped by generic COM code.

The COM server wrapper is an apartment-threading version that guarantees safe threading regardless of the client's implementation. There is a separate apartment thread for each class factory. COM objects reside on the same apartment thread as the class factory that creates them.

The `IHugs` interface functions invoke the corresponding Hugs server functions on client's behalf and pass the return values back to the client.

Invocation

The program is usually started by the COM subsystem on behalf of the client connecting to it. In this case the program is started using the standard switch `-Embedding` or `/Embedding` which means that the server will remain hidden. As an out-of-process server, `HugsSrv` is not designed to run as a standalone application and it will not start without a command-line switch.

Registering the Program

The program is automatically registered using the `register` utility if the make process completes successfully.

The program can be invoked with command line switches telling it to register on unregister itself.

To register the program, enter:

```
HugsSrv -RegServer
```
or
```
HugsSrv /RegServer
```

To unregister the program, enter:

```
HugsSrv -UnregServer
```
or
```
HugsSrv /UnregServer
```

Source Files

The source files are located in the `HugsSrv` directory.

Source files modifiable by the developer:

`makefile` the make file

`HugsSrv.h` the include file for containing class declarations, function prototypes, and resource identifiers

`HugsSrv.cpp` the main implementation file, containing the `WinMain` and `CMainWindow` implementation, as well as the main menu dispatching
`HugsSrv.rc` the resource definition file for `HugsSrv.exe`
`HugsSrv.ico` the icon resource file for `HugsSrv.exe`
`CHugs.h` the include file for the `CHugs` COM object class
`CHugs.cpp` the implementation file for the `CHugs` COM object class
`CHugsU.h` the include file for the `CHugsU` COM object class
`CHugsU.cpp` the implementation file for the `CHugsU` COM object class
`CServer.h` the include file for the server control C++ object, also used for `HugsSrv` externs
`CServer.cpp` the implementation file for the server control object, managing apartment threads, object counts, server lifetime, and the creation of class factories
`CFactory.h` the include file for the server's class factory COM objects
`CFactory.cpp` the implementation file for the server's class factories

Support for the `IHugsU` interface

In all Hugs COM server source files, the code supporting the `IHugsU` interface is compiled conditionally if the `CHUGSU_INTERFACE` macro is defined at the beginning of each file.

Hugs Server Include File (`HugsSrv.h`)

This include file defines the main window class as well as the resource identifiers (commands, strings, etc.). The window class is a class derived from the `CvirWindow` in the `AppUtil` module:
`class CMainWindow: public CvirWindow;`

The server needs to have a window in order to have a message queue for the window.

Hugs Server Source File (`HugsSrv.cpp`)

This source file contains the core of the Hugs Server functionality and implements the main framework for a Win32 application.

The `WinMain` Function

The `WinMain` function initializes the application and the COM libraries. It checks for the command line switches and if necessary, calls the appropriate registry manipulation routine.

Logging the Messages

If the server during startup can find the client window, it will be sending the log messages to the client (simplified):

```
hWnd = FindWindow(NULL, TEXT(CLIENT_WINDOW_TITLE));
if (NULL != hWnd)
{
  m_pMsgLog->LogToServer(FALSE);
  m_pMsgLog->SetClient(m_hInst, m_hWnd, hWnd);
}
else
{
  m_pMsgLog->LogToServer(TRUE);
  ::ShowWindow(m_hWnd, nShow);
  ::UpdateWindow(m_hWnd);

}
```

The CHugs COM Object Include File (CHugs.h)

This include file contains the definition of an aggregatable COHugs object.

The COHugs object demonstrates the standard way in which a COM object is defined (simplified):

```
class COHugs : public IUnknown
{
  public:
    COHugs(IUnknown* pUnkOuter, CServer* pServer);
    ~COHugs(void);
    STDMETHODIMP          QueryInterface(REFIID, PPVOID);
    STDMETHODIMP_(ULONG) AddRef(void);
    STDMETHODIMP_(ULONG) Release(void);
  private:
    class CImpIHugs : public IHugs
    {
      public:
        CImpIHugs(COHugs* pBackObj, IUnknown* pUnkOuter);
        ~CImpIHugs(void);
        STDMETHODIMP          QueryInterface(REFIID, PPVOID);
        STDMETHODIMP_(ULONG) AddRef(void);
        STDMETHODIMP_(ULONG) Release(void);
        STDMETHODIMP Test(short nNum, short *pnRes);
        STDMETHODIMP CInitHugsServer(int iArgc, unsigned char* szArg);
        STDMETHODIMP CLoadProject(unsigned char* szFileName);
      private:
        ULONG         m_C;              // Method call counter
        ULONG         m_cRefI;          // Interface Ref Count
        COHugs*       m_pBackObj;       // Parent Object back pointer.
        IUnknown*     m_pUnkOuter;      // Outer unknown for Delegation.
    };

    friend CImpIHugs;
    CImpIHugs         m_ImpIHugs;
    ULONG             m_cRefs;
    IUnknown*         m_pUnkOuter;
    CServer*          m_pServer;
};
```

After the public section of the COHugs class, a nested class CImpHugs derived from the IHugs interface is defined. This class implements the functionality outlined in the pure abstract class IHugs which cannot be instantiated. Notice that the implementation class also defines the variables it needs to run.

After the definition, the implementation class CImpHugs is declared a friend of the COHugs class and it is instantiated as a contained class.

The CHugs COM Object Source File (CHugs.cpp)

This source file implements an aggregatable COHugs object.

Reference Counting

The component implements the reference counting functions in the standard way (simplified):

```
STDMETHODIMP_(ULONG) COHugs::AddRef(void)
{
  m_cRefs++;
```

```
  return m_cRefs;
}
```

The Release function artificially bumps up the reference count just before calling the `delete` function on the object to prevent reentrancy via the main object destructor. After the object is deleted, the server housing the component is notified that the global component count should be decremented.

```
STDMETHODIMP_(ULONG) COHugs::Release(void)
{
  m_cRefs--;
  if (0 == m_cRefs)
  {
    m_cRefs++;
    delete this;
    if (NULL != m_pServer)
      m_pServer->ObjectsDown();
 }
```

Implementation of `IHugs` Interface Functions

In addition to the standard COM implementation this file contains functions that actually do the work in the `IHugs` interface, such as:

```
STDMETHODIMP
COHugs::CImpIHugs::CInitHugsServer(int iArgc, unsigned char* szArgv);
STDMETHODIMP
COHugs::CImpIHugs::CLoadProject(unsigned char* szFileName);
```

Calling Hugs Functions from the COM Code

It is necessary to prototype all Hugs' functions that are going to be invoked from the COM framework. The functions must be declared as `extern "C"` functions, e.g.:

```
extern "C" void* initHugsServer(int argc, char* argv[]);
extern "C" void LoadFile(String fn);
```

The server version of Hugs declares certain functions `static` despite the fact that these functions are callable through a virtual-table like structure of function addresses. In order to be able to link to these functions from other modules, either they have to be declared as non-static, or they must be called indirectly. The preferred solution might be to declare them non-static, since a static function implies that the function is not callable from outside of the module since it is not intended to be called from outside of the module.

Accessing Hugs Variables and Structure from the COM Code

This is a relatively tricky task because of the multitude of symbols and definitions that need to be read in. The best solution is to `#include` the proper header files in the following manner:

```
extern "C"
{
  #include "..\hugs\src\prelude.h"
  #include "..\hugs\src\storage.h"
}
```

The `CHugsU` COM Object Include File (`CHugsU.h`)

This include file contains the definition of an aggregatable `COHugsU` object.

This class defines **two** implementation classes: `CImpHugs` and `CImpHugsU`. The reason for this is that the `COHugsU` object aggregates the `COHugs` object through containment. Both implementation classes

are instantiated as private members of the COHugsU object. The consequence of this is that the object actually has two different virtual function tables, each starting with the classic trio:

```
STDMETHODIMP          QueryInterface(REFIID, PPVOID);
STDMETHODIMP_(ULONG) AddRef(void);
STDMETHODIMP_(ULONG) Release(void);
```

These two virtual tables in essence duplicate the IUnknown interface. Care must be taken to handle the implementation of QueryInterface correctly. The inner component's QueryInterface is called delegating since it delegates all queries to the outer component, whatever it is.

The CHugsU COM Object Source File (CHugsU.cpp)

This source file implements an aggregatable COHugsU object which itself aggregates the COHugs object.

Implementation of the QueryInterface Function

The inner component's QueryInterface is delegating (simplified):

```
STDMETHODIMP
COHugsU::CImpIHugs::QueryInterface(REFIID riid, PPVOID ppv)
{
  return m_pUnkOuter->QueryInterface(riid, ppv);
}
```

The outer component's QueryInterface is non-delegating (simplified):

```
STDMETHODIMP
COHugsU::QueryInterface(REFIID riid, PPVOID ppv)
{
  HRESULT hr = E_NOINTERFACE;
  *ppv = NULL;
  if (IID_IUnknown == riid)
    *ppv = this;
  else if (IID_IHugs == riid)
    *ppv = &m_ImpIHugs;
  else if (IID_IHugsU == riid)
    *ppv = &m_ImpIHugsU;

  if (NULL != *ppv)
  {
    ((LPUNKNOWN)*ppv)->AddRef();
    hr = NOERROR;
  }
  return (hr);
}
```

Reference Counting

The cast of *ppv to (LPUNKNOWN) in the call to AddRef() is important since it guarantees that for any interface in the component the AddRef() belonging to the IUnknown interface will be called. This implements the component-wide reference counting as opposed to per interface reference counting.

Implementation of the Aggregated IHugs Interface Functions

The COHugsU object's implementation must include all functions implemented by COHugs. However, these functions must be implemented using delegation (simplified):

```
STDMETHODIMP
COHugsU::CImpIHugs::Test(short nNum, short* pnRes)
```

```
{
  m_pBackObj->m_pIHugs->Test(nNum, pnRes);
  return NOERROR;
}
```

The Apartment Server Include File (`CServer.h`)

This include file contains definitions of types and classes pertaining to the apartment server. The apartment server is encapsulated in the `CServer` class:

```
class CServer : public CThreaded;
```

The `CThreaded` class is defined in the `AppUtil` module. It provides exclusive access by multiple threads to objects of classes derived from it. The exclusivity of the access is implemented using standard mutex operations.

Definition of the Apartments

```
enum { NUM_APARTMENTS = 2 };
enum { APTHUGS = 0, APTHUGSU = 1 };
```

The Apartment Server Source File (`CServer.cpp`)

The `CServer.cpp` file contains implementation of the `CServer` class. This class contains methods that start individual threads in which individual components run.

Maintaining the Server Object Count and Releasing the Server

The server maintains the variables for the object count and server locks:

```
LONG       m_cObjects;     // Global Server living Object count.
LONG       m_cLocks;       // Global Server Client Lock count.
```

The functions maintaining the object count are `CServer::ObjectsUp` and `CServer::ObjectsDown`. They are defined as follows (simplified):

```
void CServer::ObjectsUp(void)
{
  if (OwnThis())
  {
    m_cObjects += 1;
    UnOwnThis();
  }
  return;
}
```

Notice how the server shuts itself down by posting a WM_CLOSE message to its message queue if both the server object count and the locks count are zero.

```
void CServer::ObjectsDown(void)
{
  if (OwnThis())
  {
    if (m_cObjects > 0)
      m_cObjects -= 1;
  if (0L == m_cObjects && 0L == m_cLocks && IsWindow(m_hWndServer))
    {
      UnOwnThis();
      PostMessage(m_hWndServer, WM_CLOSE, 0, 0L);
    }
    else
      UnOwnThis();
  }
```

```
  return;
}
```

The Class Factories Include File (`CFactory.h`)

This include file contains definitions of class factory COM objects for the `IHugs` and the `IHugsU` interfaces. Both are derived from the `IUnknown` interface. They contain a nested `IClassFactory` implementation instantiation.

Note that both class factories are defined identically since there is no code that we need to execute to specialize the component creation.

The Class Factories Source File (`CFactory.cpp`)

This source file contains the implementation of the class factory objects. The function `CFHugs::CImpIClassFactory::CreateInstance` should be modified if it is necessary to customize the component creation process.

Modifying the Code

Enabling the `IHugsU` interface

To enable support for the `IHugsU` interface:

1. Marshaling DLL: Enable support for the `IHugsU` interface
2. Hugs Client: Enable support for the `IHugsU` interface
3. All Hugs Server `*.cpp` and `*.h` files: Uncomment the `CHUGSU_INTERFACE` macro
4. Since the `IHugsU` interface inherits from `IHugs`, it may be necessary to add the new `IHugs` functions to the `IHugsU` interface

Adding a New Function to an Existing Interface

This procedure can be **only** applied during the development phase of the program when the interface was **not** published yet. To add a new function to an existing interface, a new version of the interface with a new IID must be created.

To add a new function to an existing interface

1. Marshaling DLL: Add a new function to an existing interface
2. Hugs Client: Add a new function to an existing interface
3. Component include file: Add the declaration of the function into the component's object class. For a component that is aggregated by another component, add the function definition into the aggregating component's include file as well
4. Component source file: Add the definition of the function. For a component that is aggregated by another component, add the function definition into the aggregating component's source file as well and make sure its implementation is delegating

Adding a New Interface

To add a new interface to the Hugs COM Server program:

1. Marshaling DLL: Add a new interface
2. Hugs Client: Add a new interface
3. `HugsSrv.cpp`: Add the code to register and unregister the interface in the `CMainWindow::RegisterServer` and `CMainWindow::UnregisterServer` functions
4. `Cserver.h`: Increase the number of apartments and define the name of the new apartment
5. `Cserver.h`: In the definition of the class `Cserver`, add the member variables to store a pointer to the class factory, to the apartment `init` data structure and the apartment thread ID.
6. `Cserver.cpp`: Add the code to zero the Factory and Apartment thread references in `CServer::Cserver`

7. `Cserver.cpp`: Add the code to create, initialize and start the class factories in `CServer::OpenFactories`
8. `Cserver.cpp`: Add the code to shut down the class factories and release the interfaces in `CServer::CloseFactories`
9. `Factory.h`: Add the definition of the new class factory class
10. `Factory.cpp`: Add the implementation of the new class factory
11. Create a new component include file and define the component's object class. If the component is aggregating, add the definition of the aggregated component's implementation interface and instantiate it
12. Create a new component source file and add the standard component functionality as well as the implementations of all new interface functions. If the component is aggregating, add the functions inherited from the aggregated component

Future of the Project

Future of COM

COM is just one implementation of the component model of programming. It is possible that there will be several competing technologies that will survive. However, the sheer size of the company standing behind COM makes it very probable that COM will be one of them. If COM is successfully ported to other platforms, this likelihood becomes almost a certainty.

Where Component Architectures Are Heading

It is very likely that the component model of programming is going to gain much more attention. It is a natural extension of the object oriented programming and it can have a major impact on how large applications are architectured and developed. Componentizing applications simplifies the development since reusing components is very easy.

Where Microsoft Is Going to Take COM

Componentizing most of the programs is the number one program maintenance task at Microsoft. The task is gaining importance because of two trends:

1. Programs that were originally created as standalone applications are being integrated into suites. The two biggest examples are the Office Suite and the BackOffice Suite.
2. Generic frameworks into which individual tools are plugged in are replacing collections of tools to perform certain tasks. An example of this trend is the Microsoft Management Console with its snap-in components.

COM+

COM+ is a new standard of Component Object Model. The COM+ runtime takes care of providing implementations of `IUnknown`, `IDispatch`, error information interfaces, type information interfaces, class factories, connection points, and the standard DLL entry points, when needed.

Automation and Dual Interfaces

Automation technology builds on the COM technology to allow the components to be accessed from higher-level languages such as Visual Basic. Automation component is essentially a COM component that supports the **`IDispatch`** interface. If an Automation component supports several other interfaces, it is an **ActiveX** component.

The `IDispatch` interface (**dispinterface**) makes invocation of COM functions much easier since it allows the functions to be invoked by name. If the component implements both the virtual table invocation and the `IDispatch` interface, it is said to have **dual interfaces**.

Dispinterface implements a complex mechanism that supports type checking and automatic conversion of parameters. However, the price for the flexibility is a much more difficult implementation of the components and slower performance. For in-process servers, the standard COM virtual table interface is on the order of 100 times faster than a dispinterface. For local servers, the difference is much smaller because of the overhead of marshaling, and a virtual table interface is about 2.5 times faster than a dispinterface.

Frameworks for Development of COM and Automation Components

The COM and Automation components can be developed in almost any language. However, C++ has been the language of choice for many developers since it is the language in which COM was originally developed. Many binary standards of COM directly correspond to the structures generated by a C++ compiler.

Development of COM components in C++ from ground up is a tedious and non-trivial task. It is recommended solely for components that need to implement only a very small number of (nonstandard) interfaces.

Several C++ class frameworks have been written to facilitate component development:

- Active Control Framework (ACF)
- Active Template Library (ATL)
- Microsoft Foundation Classes (MFC)

There is a lot of confusion in the developer community about these frameworks and their recommended usage.

Microsoft Foundation Classes

MFC was the very first one to support writing of ActiveX controls. MFC classes encapsulate a lot of functionality required by the ActiveX controls and make the development process a lot easier. Its drawback is that the controls have quite a big footprint and they require the correct version of the MFC runtime DLL to run. Therefore, its use is not recommended if small footprint of the component is required. Using MFC to develop ActiveX and OLE components is very well documented.

Active Control Framework

In response to the MFC, another developer group at Microsoft came up with the Active Control Framework. The controls created using this framework are extremely small since the framework provides only the basic functionality. If programmed carefully, the controls do not even have to be linked with the C++ runtime libraries, which can result in extremely small executables. However, using ACF is very difficult since requires detailed knowledge of ActiveX functionality and a lot of experience. The documentation is practically non-existent. According to several developers at Microsoft, this framework is not going to be developed any more and most likely will be abandoned completely.

Active Template Library

The third framework that can be used to develop components is the Active Template Library. It supports a lot of functionality required by the ActiveX controls but the support for ActiveX documents and higher-level components is not very good. ATL is quickly becoming the framework that Microsoft internally uses to develop their components and therefore it is evolving very rapidly. Currently it is in version 2.1. It is distributed as a part of Visual C++ together with a wizard that greatly simplifies laying out a foundation for a component.

Enhancing the Project

A good integration of Hugs and a COM server can provide many benefits for both development and execution of Hugs programs. The results achieved in this project are only the first step. The project can

serve as a good foundation for future enhancements. The COM framework was written in plain C++, therefore it is easily portable to other platforms. All code is written to the COM specification and contains support for advanced features such as reuse by aggregation.

There are a number of directions this project can be enhanced in.

Graphical User Interface for the Client

Currently, the client is essentially a simple text-oriented application that prints traces to the main window. It is easy to eliminate the traces or, even better, to redirect them to a separate window. A graphical interface could be added to the client to give it a more GUI look and feel.

Symbol Table Browser

To support out-of-process or remote browsing and debugging of Hugs programs, a lot of support must be added in terms of COM interfaces. The COM specification suggests that it is better to support many smaller interfaces then fewer big ones. For Hugs, this might translate to a separate interface for each area of functionality.

Let's suppose the goal is to have a complete symbol table browser. The Hugs server could communicate with clients using the following interfaces:

`IGenRuntime`	to support generic runtime functionality, such as initialize, load file, load project, etc.
`IBrowseModule`	to support module browsing
`IBrowseTyCon`	to support type constructor browsing
`IBrowseStack`	to support stack browsing

It is important to realize that the server has complete access to all of Hugs structures and variables, but the client does not. The consequence is that the definition of the structures passed from the server to the client must be well thought-out. The communication must be coarse-grained enough in order to achieve reasonable performance. If, for example, each integer value should be converted to string upon client's request, a lot of communication could be generated just to pass small quantities of information around.

Using ATL

ATL seems a tool of choice for internal development of COM components at Microsoft and there is a lot of effort being poured into its development. ATL is likely to encapsulate or provide support for many standard interfaces that the Hugs server may want to implement in future.

References

1. Dale Rogerson: Inside COM, Microsoft Press 1997
2. Kraig Brockschmidt: Inside OLE, 2nd edition, Microsoft Press 1995
3. Microsoft Corporation: Automation Programmer's Reference, Microsoft Press 1997